

1. algorithm

`#include <algorithm> #include <numeric>`

Algo	Params	Funcion
sort, stable_sort	f, l	ordena el intervalo
nth_element	f, nth, l	void ordena el n-esimo, y particiona el resto
fill, fill_n	f, l / n, elem	void llena [f, l) o [f, f+n) con elem
lower_bound, upper_bound	f, l, elem	it al primer / ultimo donde se puede insertar elem para que quede ordenada
binary_search	f, l, elem	bool esta elem en [f, l)
copy	f, l, resul	hace resul+i=f+i $\forall i$
find, find_if, find_first_of	f, l, elem / pred / f2, l2	it encuentra i $\in [f, l)$ tq. i=elem, pred(i), i $\in [f2, l2)$
count, count_if	f, l, elem/pred	cuenta elem, pred(i)
search	f, l, f2, l2	busca [f2, l2) $\in [f, l)$
replace, replace_if	f, l, old / pred, new	cambia old / pred(i) por new
reverse	f, l	da vuelta
partition, stable_partition	f, l, pred	pred(i) ad, !pred(i) atras
min_element, max_element	f, l, [comp]	it min, max de [f, l]
lexicographical_compare	f1, l1, f2, l2	bool con [f1, l1] i [f2, l2]
next/prev_permutation	f, l	deja en [f, l) la perm sig, ant
set_intersection, set_difference, set_union, set_symmetric_difference,	f1, l1, f2, l2, res	[res, ...) la op. de conj
push_heap, pop_heap, make_heap	f, l, e / e /	mete/saca e en heap [f, l), hace un heap de [f, l)
is_heap	f, l	bool es [f, l) un heap
accumulate	f, l, i, [op]	$T = \sum / \text{oper de } [f, l)$
inner_product	f1, l1, f2, i	$T = i + [f1, l1) \cdot [f2, \dots)$
partial_sum	f, l, r, [op]	$r+i = \sum / \text{oper de } [f, f+i) \forall i \in [f, l)$
__builtin_ffs	unsigned int	Pos. del primer 1 desde la derecha
__builtin_clz	unsigned int	Cant. de ceros desde la izquierda.
__builtin_ctz	unsigned int	Cant. de ceros desde la derecha.
__builtin_popcount	unsigned int	Cant. de 1's en x.
__builtin_parity	unsigned int	1 si x es par, 0 si es impar.
__builtin_XXXXXXll	unsigned ll	= pero para long long's.

2. Estructuras

2.1. RMQ (static)

Dado un arreglo y una operacion asociativa *idempotente*, get(i, j) opera sobre el rango [i, j). Restriccion: $LVL \geq \text{ceil}(\log n)$; Usar [] para llenar arreglo y luego build().

```

1 struct RMQ{
2     #define LVL 10
3     tipo vec[LVL] [1<<(LVL+1)];
4     tipo &operator[] (int p){return vec[0] [p];}
5     tipo get(int i, int j) {//intervalo [i,j)
6         int p = 31-__builtin_clz(j-i);
7         return min(vec[p] [i], vec[p] [j-(1<<p)]);
8     }
9     void build(int n) {//O(nlogn)
10        int mp = 31-__builtin_clz(n);
11        forn(p, mp) forn(x, n-(1<<p))
12            vec[p+1] [x] = min(vec[p] [x], vec[p] [x+(1<<p)]);
13    }
};

```

2.2. RMQ (dynamic and lazy)

```

1 //Dado un arreglo y una operacion asociativa con neutro, get(i, j) opera
   sobre el rango [i, j).
2 typedef int Elem; //Elem de los elementos del arreglo
3 typedef int Alt; //Elem de la alteracion/**
4 #define operacion(x,y) x+y
5 const Elem neutro=0; const Alt neutro2=0;/**
6 #define MAXN 100000
7 struct RMQ{
8     int sz;
9     Elem t[4*MAXN];
10    Alt dirty[4*MAXN]; //las alteraciones pueden ser de distinto Elem/**
11    Elem &operator[] (int p){return t[sz+p];}
12    void init(int n){//O(nlgn)
13        sz = 1 << (32-__builtin_clz(n));
14        forn(i, 2*sz) t[i]=neutro;
15        forn(i, 2*sz) dirty[i]=neutro2;/**
16    }
17    void updall(){//O(n)//para lazy limpiar dirty!
18        dfor(i, sz) t[i]=operacion(t[2*i], t[2*i+1]);}
19    void push(int n, int a, int b){//propaga el dirty a sus hijos/**

```

```

20 if(dirty[n]!=0){
21     t[n]+=dirty[n]*(b-a); //altera el nodo
22     if(n<sz){
23         dirty[2*n]+=dirty[n];
24         dirty[2*n+1]+=dirty[n];
25     }
26     dirty[n]=0;
27 }
28 }
29 Elem get(int i, int j, int n, int a, int b){ //0(lgn)
30     if(j<=a || i>=b) return neutro;
31     push(n, a, b); //corrige el valor antes de usarlo/**
32     if(i<=a && b<=j) return t[n];
33     int c=(a+b)/2;
34     return operacion(get(i, j, 2*n, a, c), get(i, j, 2*n+1, c, b));
35 }
36 Elem get(int i, int j){ return get(i, j, 1, 0, sz); }
37 //altera los valores en [i, j) con una alteracion de val
38 void alterar(Alt val, int i, int j, int n, int a, int b){ //0(lgn)**
39     push(n, a, b);
40     if(j<=a || i>=b) return;
41     if(i<=a && b<=j){
42         dirty[n]+=val;
43         push(n, a, b);
44         return;
45     }
46     int c=(a+b)/2;
47     alterar(val, i, j, 2*n, a, c), alterar(val, i, j, 2*n+1, c, b);
48     t[n]=operacion(t[2*n], t[2*n+1]); //por esto es el push de arriba
49 }
50 void alterar(Alt val, int i, int j){ alterar(val, i, j, 1, 0, sz); } /**
51 void set(int p, tipo val){ //0(lgn) //No usar con lazy!!
52     for(p+=sz; p>0 && t[p]!=val;){
53         t[p]=val;
54         p/=2;
55         val=operacion(t[p*2], t[p*2+1]);
56     }
57 }
58 }rmq;

```

2.3. Fenwick Tree

```

1 //For 2D threat each column as a Fenwick tree, by adding a nested for in
  each operation
2 struct Fenwick{
3     static const int sz=1000001;
4     tipo t[sz];
5     void adjust(int p, tipo v){ //valid with p in [1, sz), 0(lgn)
6         for(; p<sz; p+=(p&-p)) t[p]+=v; }
7     tipo sum(int p){ //cumulative sum in [1, p], 0(lgn)
8         tipo s=0;
9         for(; p; p=(p&-p)) s+=t[p];
10        return s;
11    }
12    tipo sum(int a, int b){ return sum(b)-sum(a-1); }
13    //get largest value with cumulative sum less than or equal to x;
14    //for smallest, pass x-1 and add 1 to result
15    int getind(tipo x) { //0(lgn)
16        int idx = 0, mask = N;
17        while(mask && idx < N) {
18            int t = idx + mask;
19            if(x >= tree[t])
20                idx = t, x -= tree[t];
21            mask >>= 1;
22        }
23        return idx;
24    }
};

```

2.4. Union Find

```

1 struct UnionFind{
2     vector<int> f; //the array contains the parent of each node
3     void init(int n){ f.clear(); f.insert(f.begin(), n, -1); }
4     int comp(int x){ return (f[x]==-1?x:f[x]=comp(f[x])); } //0(1)
5     bool join(int i, int j) {
6         bool con=comp(i)==comp(j);
7         if(!con) f[comp(i)] = comp(j);
8         return con;
9     }
};

```

2.5. RMQ (2D)

```

1 struct RMQ2D{
2     static const int sz=1024;
3     RMQ t[sz];
4     RMQ &operator[] (int p){ return t[sz/2+p]; }

```

```

5 void build(int n, int m){//0(nm)
6     forr(y, sz/2, sz/2+m)
7         t[y].build(m);
8     forr(y, sz/2+m, sz)
9         forn(x, sz)
10            t[y].t[x]=0;
11     dforn(y, sz/2)
12         forn(x, sz)
13            t[y].t[x]=max(t[y*2].t[x], t[y*2+1].t[x]);
14 }
15 void set(int x, int y, tipo v){//0(lgm.lgn)
16     y+=sz/2;
17     t[y].set(x, v);
18     while(y/=2)
19         t[y].set(x, max(t[y*2][x], t[y*2+1][x]));
20 }
21 //0(lgm.lgn)
22 int get(int x1, int y1, int x2, int y2, int n=1, int a=0, int b=sz/2){
23     if(y2<=a || y1>=b) return 0;
24     if(y1<=a && b<=y2) return t[n].get(x1, x2);
25     int c=(a+b)/2;
26     return max(get(x1, y1, x2, y2, 2*n, a, c),
27               get(x1, y1, x2, y2, 2*n+1, c, b));
28 }
29 };
30 //Example to initialize a grid of M rows and N columns:
31 RMQ2D rmq;
32 forn(i, M)
33     forn(j, N)
34         cin >> rmq[i][j];
35 rmq.build(N, M);

```

2.6. Big Int

```

1 #define BASEXP 6
2 #define BASE 1000000
3 #define LMAX 1000
4 struct bint{
5     int l;
6     ll n[LMAX];
7     bint(ll x=0){
8         l=1;
9         forn(i, LMAX){

```

```

10         if (x) l=i+1;
11         n[i]=x%BASE;
12         x/=BASE;
13
14     }
15 }
16 bint(string x){
17     l=(x.size()-1)/BASEXP+1;
18     fill(n, n+LMAX, 0);
19     ll r=1;
20     forn(i, sz(x)){
21         n[i / BASEXP] += r * (x[x.size()-1-i]-'0');
22         r*=10; if(r==BASE)r=1;
23     }
24 }
25 void out(){
26     cout << n[l-1];
27     dforn(i, l-1) printf("%6.6llu", n[i]);//6=BASEXP!
28 }
29 void invar(){
30     fill(n+1, n+LMAX, 0);
31     while(l>1 && !n[l-1]) l--;
32 }
33 };
34 bint operator+(const bint&a, const bint&b){
35     bint c;
36     c.l = max(a.l, b.l);
37     ll q = 0;
38     forn(i, c.l) q += a.n[i]+b.n[i], c.n[i]=q %BASE, q/=BASE;
39     if(q) c.n[c.l++] = q;
40     c.invar();
41     return c;
42 }
43 pair<bint, bool> lresta(const bint& a, const bint& b) // c = a - b
44 {
45     bint c;
46     c.l = max(a.l, b.l);
47     ll q = 0;
48     forn(i, c.l) q += a.n[i]-b.n[i], c.n[i]=(q+BASE) %BASE, q=(q+BASE)/
49         BASE-1;
50     c.invar();
51     return make_pair(c, !q);
52 }

```

```

52 bint& operator-= (bint& a, const bint& b){return a=lresta(a, b).first;}
53 bint operator- (const bint&a, const bint&b){return lresta(a, b).first;}
54 bool operator< (const bint&a, const bint&b){return !lresta(a, b).second;
    ;}
55 bool operator<= (const bint&a, const bint&b){return lresta(b, a).second;
    ;}
56 bool operator==(const bint&a, const bint&b){return a <= b && b <= a;}
57 bint operator*(const bint&a, ll b){
58     bint c;
59     ll q = 0;
60     forn(i, a.l) q += a.n[i]*b, c.n[i] = q %BASE, q/=BASE;
61     c.l = a.l;
62     while(q) c.n[c.l++] = q %BASE, q/=BASE;
63     c.invar();
64     return c;
65 }
66 bint operator*(const bint&a, const bint&b){
67     bint c;
68     c.l = a.l+b.l;
69     fill(c.n, c.n+b.l, 0);
70     forn(i, a.l){
71         ll q = 0;
72         forn(j, b.l) q += a.n[i]*b.n[j]+c.n[i+j], c.n[i+j] = q %BASE, q
            /=BASE;
73         c.n[i+b.l] = q;
74     }
75     c.invar();
76     return c;
77 }
78 pair<bint, ll> ldiv(const bint& a, ll b){// c = a / b ; rm = a % b
79     bint c;
80     ll rm = 0;
81     dforn(i, a.l){
82         rm = rm * BASE + a.n[i];
83         c.n[i] = rm / b;
84         rm %= b;
85     }
86     c.l = a.l;
87     c.invar();
88     return make_pair(c, rm);
89 }
90 bint operator/(const bint&a, ll b){return ldiv(a, b).first;}
91 ll operator%(const bint&a, ll b){return ldiv(a, b).second;}

```

```

92 pair<bint, bint> ldiv(const bint& a, const bint& b){
93     bint c;
94     bint rm = 0;
95     dforn(i, a.l){
96         if (rm.l==1 && !rm.n[0])
97             rm.n[0] = a.n[i];
98         else{
99             dforn(j, rm.l) rm.n[j+1] = rm.n[j];
100             rm.n[0] = a.n[i];
101             rm.l++;
102         }
103         ll q = rm.n[b.l] * BASE + rm.n[b.l-1];
104         ll u = q / (b.n[b.l-1] + 1);
105         ll v = q / b.n[b.l-1] + 1;
106         while (u < v-1){
107             ll m = (u+v)/2;
108             if (b*m <= rm) u = m;
109             else v = m;
110         }
111         c.n[i]=u;
112         rm-=b*u;
113     }
114     c.l=a.l;
115     c.invar();
116     return make_pair(c, rm);
117 }
118 bint operator/(const bint&a, const bint&b){return ldiv(a, b).first;}
119 bint operator%(const bint&a, const bint&b){return ldiv(a, b).second;}

```

2.7. Modnum

```

1 struct mnum{
2     static const tipo mod=12582917;
3     tipo v;
4     mnum(tipo v=0): v(v%mod) {}
5     mnum operator+(mnum b){return v+b.v;}
6     mnum operator-(mnum b){return v>=b.v? v-b.v : mod-b.v+v;}
7     mnum operator*(mnum b){return v*b.v;}
8     mnum operator^(int n){
9         if(!n) return 1;
10        return n%2? (*this)^(n/2)*(*this) : (*this)^(n/2);}
11 };

```

2.8. Treap

```

1  typedef int Key;
2  typedef struct node *pnode;
3  struct node{
4      Key key;
5      int prior, size;
6      pnode l,r;
7      node(Key key=0, int prior=0): key(key), prior(prior), size(1), l(0),
          r(0) {}
8  };
9  struct treap {
10     pnode root;
11     treap(): root(0) {}
12     int size(pnode p) { return p ? p->size : 0; }
13     int size() { return size(root); }
14     void push(pnode p) {
15         // modificar y propagar el dirty a los hijos aca(para lazy)
16     }
17     // Update function and size from children's values
18     void pull(pnode p) { //recalcular valor del nodo aca (para rmq)
19         p->size = 1 + size(p->l) + size(p->r);
20     }
21     pnode merge(pnode l, pnode r) {
22         if (!l || !r) return l ? l : r;
23         push(l), push(r);
24         pnode t;
25         if (l->prior < r->prior) l->r=merge(l->r, r), t = l;
26         else r->l=merge(l, r->l), t = r;
27         pull(t);
28         return t;
29     } //opcional:
30     void merge(treap t) {root = merge(root, t.root), t.root=0;}
31     //*****KEY OPERATIONS*****//
32     void splitKey(pnode t, Key key, pnode &l, pnode &r) {
33         if (!t) return void(l = r = 0);
34         push(t);
35         if (key <= t->key) splitKey(t->l, key, l, t->l), r = t;
36         else splitKey(t->r, key, t->r, r), l = t;
37         pull(t);
38     }
39     void insertKey(Key key) {
40         pnode elem = new node(key, rand());

```

```

41         pnode t1, t2; splitKey(root, key, t1, t2);
42         t1=merge(t1,elem);
43         root=merge(t1,t2);
44     }
45     void eraseKeys(Key key1, Key key2) {
46         pnode t1,t2,t3;
47         splitKey(root,key1,t1,t2);
48         splitKey(t2,key2, t2, t3);
49         root=merge(t1,t3);
50     }
51     void eraseKey(pnode &t, Key key) {
52         if (!t) return;
53         push(t);
54         if (key == t->key) t=merge(t->l, t->r);
55         else if (key < t->key) eraseKey(t->l, key);
56         else eraseKey(t->r, key);
57         pull(t);
58     }
59     void eraseKey(Key key) {eraseKey(root, key);}
60     pnode findKey(pnode t, Key key) {
61         if (!t) return 0;
62         if (key == t->key) return t;
63         if (key < t->key) return findKey(t->l, key);
64         return findKey(t->r, key);
65     }
66     pnode findKey(Key key) { return findKey(root, key); }
67     //*****POS OPERATIONS*****// No mezclar con las funciones Key
68     //(No funciona con pos:)
69     void splitSize(pnode t, int sz, pnode &l, pnode &r) {
70         if (!t) return void(l = r = 0);
71         push(t);
72         if (sz <= size(t->l)) splitSize(t->l, sz, l, t->l), r = t;
73         else splitSize(t->r, sz - 1 - size(t->l), t->r, r), l = t;
74         pull(t);
75     }
76     void insertPos(int pos, Key key) {
77         pnode elem = new node(key, rand());
78         pnode t1,t2; splitSize(root, pos, t1, t2);
79         t1=merge(t1,elem);
80         root=merge(t1,t2);
81     }
82     void erasePos(int pos1, int pos2=-1) {
83         if(pos2==-1) pos2=pos1+1;

```

```

84     pnode t1,t2,t3;
85     splitSize(root,pos1,t1,t2);
86     splitSize(t2,pos2-pos1,t2,t3);
87     root=merge(t1, t2);
88 }
89 pnode findPos(pnode t, int pos) {
90     if(!t) return 0;
91     if(pos <= size(t->l)) return findPos(t->l, pos);
92     return findPos(t->r, pos - 1 - size(t->l));
93 }
94 Key &operator[](int pos){return findPos(root, pos)->key;}//ojito
95 };

```

3. Algos

3.1. Longest Increasing Subsequence

```

1 //Para non-increasing, cambiar comparaciones y revisar busq binaria
2 //Given an array, paint it in the least number of colors so that each
   color turns to a non-increasing subsequence.
3 //Solution:Min number of colors=Length of the longest increasing
   subsequence
4 int N, a[MAXN]; //secuencia y su longitud
5 ii d[MAXN+1]; //d[i]=ultimo valor de la subsecuencia de tamaño i
6 int p[MAXN]; //padres
7 vector<int> R; //respuesta
8 void rec(int i){
9     if(i== -1) return;
10    R.push_back(a[i]);
11    rec(p[i]);
12 }
13 int lis(){//O(nlogn)
14     d[0] = ii(-INF, -1); for(i, N) d[i+1]=ii(INF, -1);
15     for(i, N){
16         int j = upper_bound(d, d+N+1, ii(a[i], INF))-d;
17         if (d[j-1].first < a[i]&&a[i] < d[j].first){
18             p[i]=d[j-1].second;
19             d[j] = ii(a[i], i);
20         }
21     }
22     R.clear();
23     dforn(i, N+1) if(d[i].first!=INF){
24         rec(d[i].second); //reconstruir

```

```

25     reverse(R.begin(), R.end());
26     return i; //longitud
27 }
28 return 0;
29 }

```

3.2. Manacher

```

1 int d1[MAXN]; //d1[i]=long del maximo palindromo impar con centro en i
2 int d2[MAXN]; //d2[i]=analogo pero para longitud par
3 //0 1 2 3 4
4 //a a b c c <--d1[2]=3
5 //a a b b <--d2[2]=2 (estan uno antes)
6 void manacher(){
7     int l=0, r=-1, n=sz(s);
8     for(i, n){
9         int k=(i>r? 1 : min(d1[l+r-i], r-i));
10        while(i+k<n && i-k>=0 && s[i+k]==s[i-k]) ++k;
11        d1[i] = k--;
12        if(i+k > r) l=i-k, r=i+k;
13    }
14    l=0, r=-1;
15    for(i, n){
16        int k=(i>r? 0 : min(d2[l+r-i+1], r-i+1))+1;
17        while(i+k-1<n && i-k>=0 && s[i+k-1]==s[i-k]) k++;
18        d2[i] = --k;
19        if(i+k-1 > r) l=i-k, r=i+k-1;
20    }

```

3.3. Alpha-Beta pruning

```

1 ll alphabeta(State &s, bool player = true, int depth = 1e9, ll alpha = -
   INF, ll beta = INF) { //player = true -> Maximiza
2     if(s.isFinal()) return s.score;
3     //~ if (!depth) return s.heuristic();
4     vector<State> children;
5     s.expand(player, children);
6     int n = children.size();
7     for(i, n) {
8         ll v = alphabeta(children[i], !player, depth-1, alpha, beta);
9         if(!player) alpha = max(alpha, v);
10        else beta = min(beta, v);
11        if(beta <= alpha) break;
12    }

```

```
13 |     return !player ? alpha : beta;}

```

4. Strings

4.1. KMP

```
1 | string T;//cadena donde buscar(what)
2 | string P;//cadena a buscar(what)
3 | int b[MAXLEN]; //back table
4 | void kmppre(){//by gabina with love
5 |     int i=0, j=-1; b[0]=-1;
6 |     while(i<sz(P)){
7 |         while(j>=0 && P[i] != P[j]) j=b[j];
8 |         i++, j++;
9 |         b[i] = j;
10 |    }
11 | }
12 |
13 | void kmp(){
14 |     int i=0, j=0;
15 |     while(i<sz(T)){
16 |         while(j>=0 && T[i]!=P[j]) j=b[j];
17 |         i++, j++;
18 |         if(j==sz(P)){
19 |             printf("P is found at index %d in T\n", i-j);
20 |             j=b[j];
21 |         }
22 |     }
23 | }
```

4.2. Trie

```
1 | struct trie{
2 |     map<char, trie> m;
3 |     void add(const string &s, int p=0){
4 |         if(s[p]) m[s[p]].add(s, p+1);
5 |     }
6 |     void dfs(){
7 |         //Do stuff
8 |         forall(it, m)
9 |             it->second.dfs();
10 |    }
11 |};
```

4.3. Suffix Array (largo, nlogn)

```
1 | #define MAX_N 1000
2 | #define rBOUND(x) (x<n? r[x] : 0)
3 | //sa will hold the suffixes in order.
4 | int sa[MAX_N], r[MAX_N], n;
5 | string s; //input string, n=sz(s)
6 |
7 | int f[MAX_N], tmpsa[MAX_N];
8 | void countingSort(int k){
9 |     zero(f);
10 |    forn(i, n) f[rBOUND(i+k)]++;
11 |    int sum=0;
12 |    forn(i, max(255, n)){
13 |        int t=f[i]; f[i]=sum; sum+=t;}
14 |    forn(i, n)
15 |        tmpsa[f[rBOUND(sa[i]+k)]++] = sa[i];
16 |    memcpy(sa, tmpsa, sizeof(sa));
17 | }
18 | void constructsA(){//O(n log n)
19 |     n=sz(s);
20 |     forn(i, n) sa[i]=i, r[i]=s[i];
21 |     for(int k=1; k<n; k<=1){
22 |         countingSort(k), countingSort(0);
23 |         int rank, tmpr[MAX_N];
24 |         tmpr[sa[0]]=rank=0;
25 |         forr(i, 1, n)
26 |             tmpr[sa[i]]=(r[sa[i]]==r[sa[i-1]] && r[sa[i]+k]==r[sa[i-1]+k])?
27 |                 rank : ++rank;
28 |         memcpy(r, tmpr, sizeof(r));
29 |         if(r[sa[n-1]]==n-1) break;
30 |     }
31 | }
32 | void print(){//for debug
33 |     forn(i, n)
34 |         cout << i << ' ' <<
35 |         s.substr(sa[i], s.find('$', sa[i])-sa[i]) << endl;}
```

4.4. String Matching With Suffix Array

```
1 | //returns (lowerbound, upperbound) of the search
2 | ii stringMatching(string P){ //O(sz(P)lgn)
3 |     int lo=0, hi=n-1, mid=lo;
```



```

4   while(lo<hi){
5       mid=(lo+hi)/2;
6       int res=s.compare(sa[mid], sz(P), P);
7       if(res>=0) hi=mid;
8       else lo=mid+1;
9   }
10  if(s.compare(sa[lo], sz(P), P)!=0) return ii(-1, -1);
11  ii ans; ans.fst=lo;
12  lo=0, hi=n-1, mid;
13  while(lo<hi){
14      mid=(lo+hi)/2;
15      int res=s.compare(sa[mid], sz(P), P);
16      if(res>0) hi=mid;
17      else lo=mid+1;
18  }
19  if(s.compare(sa[hi], sz(P), P)!=0) hi--;
20  ans.snd=hi;
21  return ans;
22 }

1  //Calculates the LCP between consecutives suffixes in the Suffix Array.
2  //LCP[i] is the length of the LCP between sa[i] and sa[i-1]
3  int LCP[MAX_N], phi[MAX_N], PLCP[MAX_N];
4  void computeLCP(){//O(n)
5      phi[sa[0]]=-1;
6      forr(i, 1, n) phi[sa[i]]=sa[i-1];
7      int L=0;
8      forn(i, n){
9          if(phi[i]==-1) {PLCP[i]=0; continue;}
10         while(s[i+L]==s[phi[i]+L]) L++;
11         PLCP[i]=L;
12         L=max(L-1, 0);
13     }
14     forn(i, n) LCP[i]=PLCP[sa[i]];
15 }

```

4.5. Corasick

```

1  struct trie{
2      map<char, trie> next;
3      trie* tran[256]; //transiciones del automata
4      int idhoja, szhoja; //id de la hoja o 0 si no lo es

```

```

6  //link lleva al sufijo mas largo, nxthoja lleva al mas largo pero que
   es hoja
7  trie *padre, *link, *nxthoja;
8  char pch; //caracter que conecta con padre
9  trie(): tran(), idhoja(), padre(), link() {}
10 void insert(const string &s, int id=1, int p=0){ //id>0!!!
11     if(p<sz(s)){
12         trie &ch=next[s[p]];
13         tran[(int)s[p]]=&ch;
14         ch.padre=this, ch.pch=s[p];
15         ch.insert(s, id, p+1);
16     }
17     else idhoja=id, szhoja=sz(s);
18 }
19 trie* get_link() {
20     if(!link){
21         if(!padre) link=this; //es la raiz
22         else if(!padre->padre) link=padre; //hijo de la raiz
23         else link=padre->get_link()->get_tran(pch);
24     }
25     return link;
26 }
27 trie* get_tran(int c) {
28     if(!tran[c])
29         tran[c] = !padre? this : this->get_link()->get_tran(c);
30     return tran[c];
31 }
32 trie *get_nxthoja(){
33     if(!nxthoja) nxthoja = get_link()->idhoja? link : link->nxthoja;
34     return nxthoja;
35 }
36 void print(int p){
37     if(idhoja)
38         cout << "found_" << idhoja << "_at_position_" << p-szhoja << endl
39         ;
40     if(get_nxthoja()) get_nxthoja()->print(p);
41 }
42 void matching(const string &s, int p=0){
43     print(p);
44     if(p<sz(s)) get_tran(s[p])->matching(s, p+1);

```


5. Geometria

5.1. Punto

```

1 struct pto{
2     tipo x, y;
3     pto(tipo x=0, tipo y=0):x(x),y(y){}
4     pto operator+(pto a){return pto(x+a.x, y+a.y);}
5     pto operator-(pto a){return pto(x-a.x, y-a.y);}
6     pto operator+(tipo a){return pto(x+a, y+a);}
7     pto operator*(tipo a){return pto(x*a, y*a);}
8     pto operator/(tipo a){return pto(x/a, y/a);}
9     //dot product, producto interno:
10    tipo operator*(pto a){return x*a.x+y*a.y;}
11    //module of the cross product or vectorial product:
12    //if a is less than 180 clockwise from b, a^b>0
13    tipo operator^(pto a){return x*a.y-y*a.x;}
14    //returns true if this is at the left side of line qr
15    bool left(pto q, pto r){return ((q-*this)^(r-*this))>0;}
16    bool operator<(const pto &a) const{return x<a.x || (abs(x-a.x)<EPS &&
        y<a.y);}
17    bool operator==(pto a){return abs(x-a.x)<EPS && abs(y-a.y)<EPS;}
18    double norm(){return sqrt(x*x+y*y);}
19    tipo norm_sq(){return x*x+y*y;}
20 };
21 double dist(pto a, pto b){return (b-a).norm();}
22 typedef pto vec;
23
24 double angle(pto a, pto o, pto b){
25     pto oa=a-o, ob=b-o;
26     return atan2(oa^ob, oa*ob);}
27
28 //rotate p by theta rads CCW w.r.t. origin (0,0)
29 pto rotate(pto p, double theta){
30     return pto(p.x*cos(theta)-p.y*sin(theta),
31         p.x*sin(theta)+p.y*cos(theta));
32 }
33
34 //orden total de puntos alrededor de un punto r
35 struct Cmp{
36     pto r;
37     Cmp(pto _r){r = _r;}
38     int cuad(const pto &a) const{

```

```

39         if(a.x > 0 && a.y >= 0)return 0;
40         if(a.x <= 0 && a.y > 0)return 1;
41         if(a.x < 0 && a.y <= 0)return 2;
42         if(a.x >= 0 && a.y < 0)return 3;
43         assert(a.x ==0 && a.y==0);
44         return -1;
45     }
46     bool cmp(const pto&p1, const pto&p2)const{
47         int c1 = cuad(p1), c2 = cuad(p2);
48         if(c1==c2){
49             return p1.y*p2.x<p1.x*p2.y;
50         }else{
51             return c1 < c2;
52         }
53     }
54     bool operator()(const pto&p1, const pto&p2) const{
55         return cmp(pto(p1.x-r.x,p1.y-r.y),pto(p2.x-r.x,p2.y-r.y));
56     };

```

5.2. Line

```

1 struct line{
2     line() {}
3     double a,b,c;//Ax+By=C
4     //pto MUST store float coordinates!
5     line(double a, double b, double c):a(a),b(b),c(c){}
6     line(pto p, pto q): a(q.y-p.y), b(p.x-q.x), c(a*p.x+b*p.y) {}
7 };
8 bool parallels(line l1, line l2){return abs(l1.a*l2.b-l2.a*l1.b)<EPS;}
9 pto inter(line l1, line l2){//intersection
10     double det=l1.a*l2.b-l2.a*l1.b;
11     if(abs(det)<EPS) return pto(INF, INF);//parallels
12     return pto((l2.b*l1.c-l1.b*l2.c, l1.a*l2.c-l2.a*l1.c)/det;
13 }

```

5.3. Segment

```

1 struct segm{
2     pto s,f;
3     segm(pto s, pto f):s(s), f(f) {}
4     pto closest(pto p) {//use for dist to point
5         double l2 = dist_sq(s, f);
6         if(l2==0.) return s;
7         double t =((p-s)*(f-s))/l2;

```

```

8     if (t<0.) return s;//not write if is a line
9     else if(t>1.)return f;//not write if is a line
10    return s+((f-s)*t);
11    }
12    bool inside(pto p){
13    return ((s-p)^(f-p))==0 && min(s, f)< *this && *this<max(s, f);}
14    };
15
16    bool insidebox(pto a, pto b, pto p) {
17        return (a.x-p.x)*(p.x-b.x)>-EPS && (a.y-p.y)*(p.y-b.y)>-EPS;
18    }
19    pto inter(segm s1, segm s2){
20        pto r=inter(line(s1.s, s1.f), line(s2.s, s2.f));
21        if(insidebox(s1.s,s1.f,p) && insidebox(s2.s,s2.f,p))
22            return r;
23        return pto(INF, INF);
24    }

```

5.4. Polygon Area

```

1 double area(vector<pto> &p){//0(sz(p))
2     double area=0;
3     forn(i, sz(p)) area+=p[i]^p[(i+1)%sz(p)];
4     //if points are in clockwise order then area is negative
5     return abs(area)/2;
6 }
7 //Area ellipse = M_PI*a*b where a and b are the semi axis lengths
8 //Area triangle = sqrt(s*(s-a)(s-b)(s-c)) where s=(a+b+c)/2

```

5.5. Circle

```

1 vec perp(vec v){return vec(-v.y, v.x);}
2 line bisector(pto x, pto y){
3     line l=line(x, y); pto m=(x+y)/2;
4     return line(-l.b, l.a, -l.b*m.x+l.a*m.y);
5 }
6 struct Circle{
7     pto o;
8     double r;
9     Circle(pto x, pto y, pto z){
10         o=inter(bisector(x, y), bisector(y, z));
11         r=dist(o, x);
12     }
13     pair<pto, pto> ptosTang(pto p){

```

```

14     pto m=(p+o)/2;
15     tipo d=dist(o, m);
16     tipo a=r*r/(2*d);
17     tipo h=sqrt(r*r-a*a);
18     pto m2=o+(m-o)*a/d;
19     vec per=perp(m-o)/d;
20     return mkp(m2-per*h, m2+per*h);
21 }
22 };
23 //finds the center of the circle containing p1 and p2 with radius r
24 //as there may be two solutions swap p1, p2 to get the other
25 bool circle2PtsRad(pto p1, pto p2, double r, pto &c){
26     double d2=(p1-p2).norm_sq(), det=r*r/d2-0.25;
27     if(det<0) return false;
28     c=(p1+p2)/2+perp(p2-p1)*sqrt(det);
29     return true;
30 }
31 #define sqr(a) ((a)*(a))
32 #define feq(a,b) (fabs((a)-(b))<EPS)
33 pair<tipo, tipo> ecCuad(tipo a, tipo b, tipo c){//a*x*x+b*x+c=0
34     tipo dx = sqrt(b*b-4.0*a*c);
35     return make_pair((-b + dx)/(2.0*a), (-b - dx)/(2.0*a));
36 }
37 pair<pto, pto> interCL(Circle c, line l){
38     bool sw=false;
39     if((sw=feq(0,l.b))){
40         swap(l.a, l.b);
41         swap(c.o.x, c.o.y);
42     }
43     pair<tipo, tipo> rc = ecCuad(
44         sqr(l.a)+sqr(l.b),
45         2.0*l.a*l.b*c.o.y-2.0*(sqr(l.b)*c.o.x+l.c*l.a),
46         sqr(l.b)*(sqr(c.o.x)+sqr(c.o.y)-sqr(c.r))+sqr(l.c)-2.0*l.c*l.b*c.o.y
47     );
48     pair<pto, pto> p( pto(rc.first, (l.c - l.a * rc.first) / l.b),
49         pto(rc.second, (l.c - l.a * rc.second) / l.b) );
50     if(sw){
51         swap(p.first.x, p.first.y);
52         swap(p.second.x, p.second.y);
53     }
54     return p;
55 }
56 pair<pto, pto> interCC(Circle c1, Circle c2){

```

```

57 | line l;
58 | l.a = c1.o.x-c2.o.x;
59 | l.b = c1.o.y-c2.o.y;
60 | l.c = (sqr(c2.r)-sqr(c1.r)+sqr(c1.o.x)-sqr(c2.o.x)+sqr(c1.o.y)
61 | -sqr(c2.o.y))/2.0;
62 | return interCL(c1, l);
63 | }

```

5.6. Point in Poly

```

1 | //checks if v is inside of P, using ray casting
2 | //works with convex and concave.
3 | //excludes boundaries, handle it separately using segment.inside()
4 | bool inPolygon(pto v, vector<pto>& P) {
5 |     bool c = false;
6 |     forn(i, sz(P)){
7 |         int j=(i+1)%sz(P);
8 |         if((P[j].y>v.y) != (P[i].y > v.y) &&
9 | (v.x < (P[i].x - P[j].x) * (v.y-P[j].y) / (P[i].y - P[j].y) + P[j].x))
10 |             c = !c;
11 |     }
12 |     return c;
13 | }

```

5.7. Convex Check CHECK

```

1 | bool isConvex(vector<int> &p){//O(N)
2 |     int N=sz(p);
3 |     if(N<3) return false;
4 |     bool isLeft=p[0].left(p[1], p[2]);
5 |     forr(i, 1, N)
6 |         if(p[i].left(p[(i+1)%N], p[(i+2)%N])!=isLeft)
7 |             return false;
8 |     return true; }

```

5.8. Convex Hull

```

1 | //stores convex hull of P in S, CCW order
2 | void CH(vector<pto>& P, vector<pto> &S){
3 |     S.clear();
4 |     sort(P.begin(), P.end());
5 |     forn(i, sz(P)){
6 |         while(sz(S)>= 2 && S[sz(S)-1].left(S[sz(S)-2], P[i])) S.pop_back();
7 |         S.pb(P[i]);

```

```

8 |     }
9 |     S.pop_back();
10 |     int k=sz(S);
11 |     dforn(i, sz(P)){
12 |         while(sz(S) >= k+2 && S[sz(S)-1].left(S[sz(S)-2], P[i])) S.pop_back
13 |             ();
14 |         S.pb(P[i]);
15 |     }
16 |     S.pop_back();

```

5.9. Cut Polygon

```

1 | //cuts polygon Q along the line ab
2 | //stores the left side (swap a, b for the right one) in P
3 | void cutPolygon(pto a, pto b, vector<pto> Q, vector<pto> &P){
4 |     P.clear();
5 |     forn(i, sz(Q)){
6 |         double left1=(b-a)^(Q[i]-a), left2=(b-a)^(Q[(i+1)%sz(Q)]-a);
7 |         if(left1>=0) P.pb(Q[i]);
8 |         if(left1*left2<0)
9 |             P.pb(inter(line(Q[i], Q[(i+1)%sz(Q)]), line(a, b)));
10 |     }
11 | }

```

5.10. Bresenham

```

1 | //plot a line approximation in a 2d map
2 | void bresenham(pto a, pto b){
3 |     pto d=b-a; d.x=abs(d.x), d.y=abs(d.y);
4 |     pto s(a.x<b.x? 1: -1, a.y<b.y? 1: -1);
5 |     int err=d.x-d.y;
6 |     while(1){
7 |         m[a.x][a.y]=1;//plot
8 |         if(a==b) break;
9 |         int e2=2*err;
10 |         if(e2 > -d.y){
11 |             err-=d.y, a.x+=s.x;
12 |         if(e2 < d.x)
13 |             err+= d.x, a.y+= s.y;
14 |         }
15 |     }

```

5.11. Interseccion de Circulos en $n^3 \log(n)$

```

1 struct event {
2     double x; int t;
3     event(double xx, int tt) : x(xx), t(tt) {}
4     bool operator <(const event &o) const { return x < o.x; }
5 };
6 typedef vector<Circle> VC;
7 typedef vector<event> VE;
8 int n;
9 double cuenta(VE &v, double A, double B) {
10     sort(v.begin(), v.end());
11     double res = 0.0, lx = ((v.empty())?0.0:v[0].x);
12     int contador = 0;
13     forn(i,sz(v)) {
14         // interseccion de todos (contador == n), union de todos (
15         // conjunto de puntos cubierto por exacta k Circulos (contador
16         // == k)
17         if (contador == n) res += v[i].x - lx;
18         contador += v[i].t;
19         lx = v[i].x;
20     }
21     return res;
22 }
23 // Primitiva de sqrt(r*r - x*x) como funcion double de una variable x.
24 inline double primitiva(double x, double r) {
25     if (x >= r) return r*r*M_PI/4.0;
26     if (x <= -r) return -r*r*M_PI/4.0;
27     double raiz = sqrt(r*r-x*x);
28     return 0.5 * (x * raiz + r*r*atan(x/raiz));
29 }
30 double interCircle(VC &v) {
31     vector<double> p; p.reserve(v.size() * (v.size() + 2));
32     forn(i,sz(v)) {
33         p.push_back(v[i].c.x + v[i].r);
34         p.push_back(v[i].c.x - v[i].r);
35     }
36     forn(i,sz(v)) forn(j,i) {
37         Circle &a = v[i], b = v[j];
38         double d = (a.c - b.c).norm();
39         if (fabs(a.r - b.r) < d && d < a.r + b.r) {
40             double alfa = acos((sqr(a.r) + sqr(d) - sqr(b.r)) / (2.0 * d

```

```

41         p.pb((a.c + rotate(vec, alfa)).x);
42         p.pb((a.c + rotate(vec, -alfa)).x);
43     }
44 }
45 sort(p.begin(), p.end());
46 double res = 0.0;
47 forn(i,sz(p)-1) {
48     const double A = p[i], B = p[i+1];
49     VE ve; ve.reserve(2 * v.size());
50     forn(j,sz(v)) {
51         const Circle &c = v[j];
52         double arco = primitiva(B-c.c.x,c.r) - primitiva(A-c.c.x,c.r
53         );
54         double base = c.c.y * (B-A);
55         ve.push_back(event(base + arco,-1));
56         ve.push_back(event(base - arco, 1));
57     }
58     res += cuenta(ve,A,B);
59 }
60 return res;
61 }

```

6. Math

6.1. Identidades

$$\begin{aligned}
 \sum_{i=0}^n \binom{n}{i} &= 2^n \\
 \sum_{i=0}^n i \binom{n}{i} &= n * 2^{n-1} \\
 \sum_{i=m}^n i &= \frac{n(n+1)}{2} - \frac{m(m-1)}{2} = \frac{(n+1-m)(n+m)}{2} \\
 \sum_{i=0}^n i &= \sum_{i=1}^n i = \frac{n(n+1)}{2} \\
 \sum_{i=0}^n i^2 &= \frac{n(n+1)(2n+1)}{6} = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} \\
 \sum_{i=0}^n i(i-1) &= \frac{8}{6} \left(\frac{n}{2}\right) \left(\frac{n}{2} + 1\right) (n+1) \text{ (doubles)} \rightarrow \text{Sino ver caso impar y par} \\
 \sum_{i=0}^n i^3 &= \left(\frac{n(n+1)}{2}\right)^2 = \frac{n^4}{4} + \frac{n^3}{2} + \frac{n^2}{4} = \left[\sum_{i=1}^n i\right]^2 \\
 \sum_{i=0}^n i^4 &= \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30} = \frac{n^5}{5} + \frac{n^4}{2} + \frac{n^3}{3} - \frac{n}{30} \\
 \sum_{i=0}^n i^p &= \frac{(n+1)^{p+1}}{p+1} + \sum_{k=1}^p \frac{B_k}{p-k+1} \binom{p}{k} (n+1)^{p-k+1} \\
 r &= e - v + k + 1
 \end{aligned}$$

Teorema de Pick: (Area, puntos interiores y puntos en el borde)

$$A = I + \frac{B}{2} - 1$$

6.2. Ec. Característica

$$a_0 T(n) + a_1 T(n-1) + \dots + a_k T(n-k) = 0$$

$$p(x) = a_0 x^k + a_1 x^{k-1} + \dots + a_k$$

Sean r_1, r_2, \dots, r_q las raíces distintas, de mult. m_1, m_2, \dots, m_q

$$T(n) = \sum_{i=1}^q \sum_{j=0}^{m_i-1} c_{ij} n^j r_i^n$$

Las constantes c_{ij} se determinan por los casos base.

6.3. Combinatorio

```

1 | forn(i, MAXN+1){ //comb[i][k]=i tomados de a k
2 |   comb[i][0]=comb[i][i]=1;
3 |   forr(k, 1, i) comb[i][k]=(comb[i-1][k]+comb[i-1][k-1])%MOD;
4 | }
5 |
6 | ll lucas (ll n, ll k, int p){ //Calcula (n,k)%p teniendo comb[p][p]
   |   precalculado.
7 |   ll aux = 1;
8 |   while (n + k){
9 |     aux = (aux * comb[n%p][k%p]) %p;
10 |    n/=p, k/=p;
11 |   }
12 |   return aux;
13 | }
```

6.4. Exp. de Numeros Mod.

```

1 | ll expmod (ll b, ll e, ll m){ //O(log b)
2 |   if(!e) return 1;
3 |   ll q= expmod(b,e/2,m); q=(q*q)%m;
4 |   return e%2? (b * q)%m : q;
5 | }
```

6.5. Exp. de Matrices y Fibonacci en log(n)

```

1 | struct M22{          // |a b|
2 |   tipo a,b,c,d; // |c d|
3 |   M22 operator*(const M22 &p) const {
4 |     return (M22){a*p.a+b*p.c, a*p.b+b*p.d, c*p.a+d*p.c,c*p.b+d*p.d};}
5 | };
6 | M22 operator^(const M22 &p, int n){
7 |   if(!n) return (M22){1, 0, 0, 1}; //identidad
8 |   M22 q=p^(n/2); q=q*q;
9 |   return n%2? p * q : q;}
```

```

10 |
11 | ll fibo(ll n){ //calcula el fibonacci enesimo
12 |   M22 mat=(M22){0, 1, 1, 1}^n;
13 |   return mat.a*f0+mat.b*f1; //f0 y f1 son los valores iniciales
14 | }
```

6.6. Teorema Chino del Resto

$$y = \sum_{j=1}^n (x_j * (\prod_{i=1, i \neq j}^n m_i)^{-1}_{m_j} * \prod_{i=1, i \neq j}^n m_i)$$

6.7. Funciones de primos

```

1 | ll numPrimeFactors (ll n){
2 |   ll rta = 0;
3 |   map<ll,ll> f=fact(n);
4 |   forall(it, f) rta += it->second;
5 |   return rta;
6 | }
7 |
8 | ll numDiffPrimeFactors (ll n){
9 |   ll rta = 0;
10 |  map<ll,ll> f=fact(n);
11 |  forall(it, f) rta += 1;
12 |  return rta;
13 | }
14 |
15 | ll sumPrimeFactors (ll n){
16 |   ll rta = 0;
17 |   map<ll,ll> f=fact(n);
18 |   forall(it, f) rta += it->first;
19 |   return rta;
20 | }
21 |
22 | ll numDiv (ll n){
23 |   ll rta = 1;
24 |   map<ll,ll> f=fact(n);
25 |   forall(it, f) rta *= (it->second + 1);
26 |   return rta;
27 | }
28 |
29 | ll sumDiv (ll n){
```

```

30 ll rta = 1;
31 map<ll,ll> f=fact(n);
32 forall(it, f) rta *= ((ll)pow((double)it->first, it->second + 1.0)-1)
    / (it->first-1);
33 return rta;
34 }
35
36 ll eulerPhi (ll n){ // con criba: O(lg n)
37     ll rta = n;
38     map<ll,ll> f=fact(n);
39     forall(it, f) rta -= rta / it->first;
40     return rta;
41 }
42
43 ll eulerPhi2 (ll n){ // O(sqrt n)
44     ll r = n;
45     forr (i,2,n+1){
46         if ((ll)i*i > n)
47             break;
48         if (n % i == 0){
49             while (n%i == 0) n/=i;
50             r -= r/i;
51         }}
52     if (n != 1)
53         r-= r/n;
54     return r;
55 }

```

6.8. Phollard's Rho (rolando)

```

1 ll mulmod (ll a, ll b, ll c) { //returns (a*b)%c, and minimize overflow
2     ll x = 0, y = a%c;
3     while (b > 0){
4         if (b % 2 == 1) x = (x+y) % c;
5         y = (y*2) % c;
6         b /= 2;
7     }
8     return x % c;
9 }
10
11 bool es_primo_prob (ll n, int a)
12 {
13     if (n == a) return true;

```

```

14 ll s = 0, d = n-1;
15 while (d % 2 == 0) s++, d/=2;
16
17 ll x = expmod(a,d,n);
18 if ((x == 1) || (x+1 == n)) return true;
19
20 forn (i, s-1){
21     x = (x*x)%n;
22     if (x == 1) return false;
23     if (x+1 == n) return true;
24 }
25 return false;
26 }
27
28 bool miller_rabin (ll n){ //devuelve true si n es primo
29     if (n == 1) return false;
30     const int ar[] = {2,3,5,7,11,13,17,19,23};
31     forn (j,9)
32         if (!es_primo_prob(n,ar[j]))
33             return false;
34     return true;
35 }
36
37 ll pollard_rho (ll n, ll c=1){
38     int i = 0, k = 2;
39     ll x = 3, y = 3;
40     if(c>=n) return -1; //FAILURE
41     //~ if(c!=1) dprint(c);
42     while (1){
43         i++;
44         x = (mulmod (x,x,n) + c) % n;
45         ll d = gcd (abs(y-x), n);
46         if(d==n) return pollard_rho(n, c+1);
47         if (d != 1) return d;
48         if (i == k) y = x, k*=2;
49     }
50 }
51
52 ll brent(ll n){
53     srand(time(NULL));
54     if (n%2 == 0)
55         return 2;
56     ll y = rand()%(n-1)+1, c = rand()%(n-1)+1, m = rand()%(n-1)+1;

```

```

57     ll g,r,q,x,k,ys;
58     g = r = q = 1;
59     while (g==1){
60         x = y;
61         forn (i,r)
62             y = ((y*y)%n+c)%n;
63         k = 0;
64         while (k<r && g==1){
65             ys = y;
66             forn (i,min(m,r-k)){
67                 y = ((y*y)%n+c)%n;
68                 q = q*(x-y)%n;
69             }
70             g = gcd(q,n);
71             k = k + m;
72         }
73         r = r*2;
74     }
75     if (g==n){
76         while (true){
77             ys = ((ys*ys)%n+c)%n;
78             g = gcd(abs(x-ys),n);
79             if (g>1)
80                 break;
81         }
82     }
83
84     return g;
85 }

```

6.9. Criba

```

1  #define MAXP 80000 //no necesariamente primo
2  int criba[MAXP+1];
3  vector<int> primos;
4  void buscarprimos(){
5      int sq=sqrt(MAXP)+1;
6      forr(p, 2, MAXP+1) if(!criba[p]){
7          primos.push_back(p);
8          if(p<=sq)
9              for(int m=p*p; m<=MAXP; m+=p)//borro los multiplos de p
10                 if(!criba[m])criba[m]=p;
11     }

```

```

12 }

```

6.10. Factorizacion

Sea $n = \prod p_i^{k_i}$, fact(n) genera un map donde a cada p_i le asocia su k_i

```

1  //factoriza bien numeros hasta MAXP^2
2  map<ll,ll> fact(ll n){ //0 (cant primos)
3      map<ll,ll> ret;
4      forall(p, primos){
5          while(!(n%p)){
6              ret[*p]++; //divisor found
7              n/=p;
8          }
9      }
10     if(n>1) ret[n]++;
11     return ret;
12 }
13
14 //factoriza bien numeros hasta MAXP
15 map<ll,ll> fact2(ll n){ //0 (lg n)
16     map<ll,ll> ret;
17     while (criba[n]){
18         ret[criba[n]]++;
19         n/=criba[n];
20     }
21     if(n>1) ret[n]++;
22     return ret;
23 }

```

6.11. GCD

```

1  tipo gcd(tipo a, tipo b){return a?gcd(b %a, a):b;}

```

6.12. Extended Euclid

```

1  void extendedEuclid (ll a, ll b){ //a * x + b * y = d
2      if (!b) { x = 1; y = 0; d = a; return;}
3      extendedEuclid (b, a%b);
4      ll x1 = y;
5      ll y1 = x - (a/b) * y;
6      x = x1; y = y1;
7  }

```


6.13. LCM

```
1 | tipo lcm(tipo a, tipo b){return a / gcd(a,b) * b;}
```

6.14. Inversos

```
1 | #define MAXMOD 15485867
2 | ll inv[MAXMOD]; //inv[i]*i=1 mod MOD
3 | void calc(int p){ //O(p)
4 |     inv[1]=1;
5 |     forr(i, 2, p) inv[i]= p-((p/i)*inv[p%i])%p;
6 | }
7 | int inverso(int x){ //O(log x)
8 |     return expmod(x, eulerphi(MOD)-2); //si mod no es primo(sacar a mano)
9 |     return expmod(x, MOD-2); //si mod es primo
10 | }
```

6.15. Simpson

```
1 | double integral(double a, double b, int n=10000) { //O(n), n=cantdiv
2 |     double area=0, h=(b-a)/n, fa=f(a), fb;
3 |     forn(i, n){
4 |         fb=f(a+h*(i+1));
5 |         area+=fa+ 4*f(a+h*(i+0.5)) +fb, fa=fb;
6 |     }
7 |     return area*h/6.; }
```

6.16. Fraction

```
1 | tipo mcd(tipo a, tipo b){return a?mcd(b%a, a):b;}
2 | struct frac{
3 |     tipo p,q;
4 |     frac(tipo p=0, tipo q=1):p(p),q(q) {norm();}
5 |     void norm(){
6 |         tipo a = mcd(p,q);
7 |         if(a) p/=a, q/=a;
8 |         else q=1;
9 |         if (q<0) q=-q, p=-p;}
10 | frac operator+(const frac& o){
11 |     tipo a = mcd(q,o.q);
12 |     return frac(p*(o.q/a)+o.p*(q/a), q*(o.q/a));}
13 | frac operator-(const frac& o){
14 |     tipo a = mcd(q,o.q);
15 |     return frac(p*(o.q/a)-o.p*(q/a), q*(o.q/a));}
```

```
16 | frac operator*(frac o){
17 |     tipo a = mcd(q,o.p), b = mcd(o.q,p);
18 |     return frac((p/b)*(o.p/a), (q/a)*(o.q/b));}
19 | frac operator/(frac o){
20 |     tipo a = mcd(q,o.q), b = mcd(o.p,p);
21 |     return frac((p/b)*(o.q/a), (q/a)*(o.p/b));}
22 | bool operator<(const frac &o) const{return p*o.q < o.p*q;}
23 | bool operator==(frac o){return p==o.p&&q==o.q;}
24 | };
```

6.17. Polinomio

```
1 | struct poly {
2 |     vector<tipo> c; //guarda los coeficientes del polinomio
3 |     poly(const vector<tipo> &c): c(c) {}
4 |     poly() {}
5 |     int gr(){ //calculates grade of the polynomial
6 |         return sz(c); }
7 |     bool isnull() {return c.empty();}
8 |     poly operator+(const poly &o) const {
9 |         int m = sz(c), n = sz(o.c);
10 |         vector<tipo> res(max(m,n));
11 |         forn(i, m) res[i] += c[i];
12 |         forn(i, n) res[i] += o.c[i];
13 |         return poly(res);
14 |     }
15 |     poly operator*(const poly &o) const {
16 |         int m = sz(c), n = sz(o.c);
17 |         vector<tipo> res(m+n-1);
18 |         forn(i, m) forn(j, n) res[i+j]+=c[i]*o.c[j];
19 |         return poly(res);
20 |     }
21 |     tipo eval(tipo v) {
22 |         tipo sum = 0;
23 |         forall(it, c) sum=sum*v + *it;
24 |         return sum;
25 |     }
26 |     //the following function generates the roots of the polynomial
27 |     //it can be easily modified to return float roots
28 |     set<tipo> roots(){
29 |         set<tipo> roots;
30 |         tipo a0 = abs(c[0]), an = abs(c[sz(c)-1]);
31 |         vector<tipo> ps,qs;
```

```

32   forr(p,1,sqrt(a0)+1) if (a0%p==0) ps.pb(p),ps.pb(a0/p);
33   forr(q,1,sqrt(an)+1) if (an%q==0) qs.pb(q),qs.pb(an/q);
34   forall(pt,ps)
35     forall(qt,qs) if ( (*pt) % (*qt)==0 ) {
36       tipo root = abs((*pt) / (*qt));
37       if (eval(root)==0) roots.insert(root);
38     }
39   return roots;
40 }
41 };
42 poly interpolate(const vector<tipo> &x, const vector<tipo> &y) {
43   int n = sz(x);
44   poly p;
45   vector<tipo> aux(2);
46   forn(i, n) {
47     double a = y[i] - p.eval(x[i]);
48     forn(j, i) a /= x[i] - x[j];
49     poly add(vector<tipo>(1, a));
50     forn(j, i) aux[0]=-x[j], aux[1]=1, add = add*aux;
51     p = p + add;
52   }
53   return p;
54 }
55 //the following functions allows parsing an expression like
56 //34+150+4*45
57 //into a polynomial(el numero en funcion de la base)
58 #define LAST(s) (sz(s)? s[sz(s)-1] : 0)
59 #define POP(s) s.erase(--s.end());
60 poly D(string &s) {
61   poly d;
62   for(int i=0; isdigit(LAST(s)); i++) d.c.push_back(LAST(s)-'0'), POP(s);
63   return d;}
64
65 poly T(string &s) {
66   poly t=D(s);
67   if (LAST(s)=='*'){POP(s); return T(s)*t;}
68   return t;
69 }
70 //main function, call this to parse
71 poly E(string &s) {
72   poly e=T(s);
73   if (LAST(s)=='+'){POP(s); return E(s)+e;}

```

```

74   return e;
75 }

```

6.18. Ec. Lineales

```

1  bool resolver_ev(Mat a, Vec y, Vec &x, Mat &ev){
2    int n = a.size(), m = n?a[0].size():0, rw = min(n, m);
3    vector<int> p; forn(i,m) p.push_back(i);
4    forn(i, rw) {
5      int uc=i, uf=i;
6      forr(f, i, n) forr(c, i, m) if(fabs(a[f][c])>fabs(a[uf][uc])) {uf=f;
7        uc=c;}
8      if (freq(a[uf][uc], 0)) { rw = i; break; }
9      forn(j, n) swap(a[j][i], a[j][uc]);
10     swap(a[i], a[uf]); swap(y[i], y[uf]); swap(p[i], p[uc]);
11     tipo inv = 1 / a[i][i]; //aca divide
12     forr(j, i+1, n) {
13       tipo v = a[j][i] * inv;
14       forr(k, i, m) a[j][k]-=v * a[i][k];
15       y[j] -= v*y[i];
16     }
17   } // rw = rango(a), aca la matriz esta triangulada
18   forr(i, rw, n) if (!freq(y[i],0)) return false; // chequeo de
19   // compatibilidad
20   x = vector<tipo>(m, 0);
21   dforn(i, rw){
22     tipo s = y[i];
23     forr(j, i+1, rw) s -= a[i][j]*x[p[j]];
24     x[p[i]] = s / a[i][i]; //aca divide
25   }
26   ev = Mat(m-rw, Vec(m, 0)); // Esta parte va SOLO si se necesita el ev
27   forn(k, m-rw) {
28     ev[k][p[k+rw]] = 1;
29     dforn(i, rw){
30       tipo s = -a[i][k+rw];
31       forr(j, i+1, rw) s -= a[i][j]*ev[k][p[j]];
32       ev[k][p[i]] = s / a[i][i]; //aca divide
33     }
34   }
35   return true;
36 }

```

7. Grafos

7.1. Dijkstra

```

1  #define INF 1e9
2  int N;
3  #define MAX_V 250001
4  vector<ii> G[MAX_V];
5  //To add an edge use
6  #define add(a, b, w) G[a].pb(mkp(w, b))
7
8  ll dijkstra(int s, int t){//O(|E| log |V|)
9      priority_queue<ii, vector<ii>, greater<ii> > Q;
10     vector<ll> dist(N, INF); vector<int> dad(N, -1);
11     Q.push(mkp(0, s)); dist[s] = 0;
12     while(sz(Q)){
13         ii p = Q.top(); Q.pop();
14         if(p.snd == t) break;
15         forall(it, G[p.snd])
16             if(dist[p.snd]+it->first < dist[it->snd]){
17                 dist[it->snd] = dist[p.snd] + it->fst;
18                 dad[it->snd] = p.snd;
19                 Q.push(mkp(dist[it->snd], it->snd));
20             }
21     }
22     return dist[t];
23     if(dist[t]<INF)//path generator
24     for(int i=t; i!=-1; i=dad[i])
25         printf("%d%c", i, (i==s?'\\n':' '));
26 }
```

7.2. Bellman-Ford

```

1  vector<ii> G[MAX_N]; //ady. list with pairs (weight, dst)
2  int dist[MAX_N];
3  void bford(int src){//O(VE)
4      dist[src]=0;
5      forn(i, N-1) forn(j, N) if(dist[j]!=INF) forall(it, G[j])
6          dist[it->snd]=min(dist[it->snd], dist[j]+it->fst);
7  }
8
9  bool hasNegCycle(){
10     forn(j, N) if(dist[j]!=INF) forall(it, G[j])
```

```

11     if(dist[it->snd]>dist[j]+it->fst) return true;
12     //inside if: all points reachable from it->snd will have -INF distance
13     (do bfs)
14     return false;
15 }
```

7.3. Floyd-Warshall

```

1  //G[i][j] contains weight of edge (i, j) or INF
2  //G[i][i]=0
3  int G[MAX_N][MAX_N];
4  void floyd(){//O(N^3)
5      forn(k, N) forn(i, N) if(G[i][k]!=INF) forn(j, N) if(G[k][j]!=INF)
6          G[i][j]=min(G[i][j], G[i][k]+G[k][j]);
7  }
8  bool inNegCycle(int v){
9      return G[v][v]<0;}
10 //checks if there's a neg. cycle in path from a to b
11 bool hasNegCycle(int a, int b){
12     forn(i, N) if(G[a][i]!=INF && G[i][i]<0 && G[i][b]!=INF)
13         return true;
14     return false;
15 }
```

7.4. Kruskal

```

1  struct Ar{int a,b,w;};
2  bool operator<(const Ar& a, const Ar &b){return a.w<b.w;}
3  vector<Ar> E;
4  ll kruskal(){
5      ll cost=0;
6      sort(E.begin(), E.end()); //ordenar aristas de menor a mayor
7      uf.init(n);
8      forall(it, E){
9          if(uf.comp(it->a)!=uf.comp(it->b)){//si no estan conectados
10             uf.unir(it->a, it->b); //conectar
11             cost+=it->w;
12         }
13     }
14     return cost;
15 }
```

7.5. Prim

```

1 bool taken[MAXN];
2 priority_queue<ii, vector<ii>, greater<ii> > pq; //min heap
3 void process(int v){
4     taken[v]=true;
5     forall(e, G[v])
6         if(!taken[e->second]) pq.push(*e);
7 }
8
9 ll prim(){
10     zero(taken);
11     process(0);
12     ll cost=0;
13     while(sz(pq)){
14         ii e=pq.top(); pq.pop();
15         if(!taken[e.second]) cost+=e.first, process(e.second);
16     }
17     return cost;
18 }

```

7.6. 2-SAT + Tarjan SCC

```

1 //We have a vertex representing a var and other for his negation.
2 //Every edge stored in G represents an implication. To add an equation
  of the form a||b, use addor(a, b)
3 //MAX=max cant var, n=cant var
4 #define addor(a, b) (G[neg(a)].pb(b), G[neg(b)].pb(a))
5 vector<int> G[MAX*2];
6 //idx[i]=index assigned in the dfs
7 //lw[i]=lowest index(closer from the root) reachable from i
8 int lw[MAX*2], idx[MAX*2], qidx;
9 stack<int> q;
10 int qcmp, cmp[MAX*2];
11 //verdad[cmp[i]]=valor de la variable i
12 bool verdad[MAX*2+1];
13
14 int neg(int x) { return x>=n? x-n : x+n;}
15 void tjn(int v){
16     lw[v]=idx[v]=++qidx;
17     q.push(v), cmp[v]=-2;
18     forall(it, G[v]){
19         if(!idx[*it] || cmp[*it]==-2){
20             if(!idx[*it]) tjn(*it);
21             lw[v]=min(lw[v], lw[*it]);

```

```

22     }
23 }
24 if(lw[v]==idx[v]){
25     qcmp++;
26     int x;
27     do{x=q.top(); q.pop(); cmp[x]=qcmp;}while(x!=v);
28     verdad[qcmp]=(cmp[neg(v)]<0);
29 }
30 }
31 //remember to CLEAR G!!!
32 bool satisf(){//O(n)
33     memset(idx, 0, sizeof(idx)), qidx=0;
34     memset(cmp, -1, sizeof(cmp)), qcmp=0;
35     forn(i, n){
36         if(!idx[i]) tjn(i);
37         if(!idx[neg(i)]) tjn(neg(i));
38     }
39     forn(i, n) if(cmp[i]==cmp[neg(i)]) return false;
40     return true;
41 }

```

7.7. Articulation Points

```

1 int N;
2 vector<int> G[1000000];
3 //V[i]=node number(if visited), L[i]= lowest V[i] reachable from i
4 int qV, V[1000000], L[1000000], P[1000000];
5 void dfs(int v, int f){
6     L[v]=V[v]=++qV;
7     forall(it, G[v])
8         if(!V[*it]){
9             dfs(*it, v);
10            L[v] = min(L[v], L[*it]);
11            P[v] += L[*it]>=V[v];
12        }
13        else if(*it!=f)
14            L[v]=min(L[v], V[*it]);
15    }
16    int cantart(){ //O(n)
17        qV=0;
18        zero(V), zero(P);
19        dfs(1, 0); P[1]--;
20        int q=0;

```

```

21     forn(i, N) if(P[i]) q++;
22 return q;
23 }

```

7.8. Comp. Biconexas y Puentes

```

1 struct edge {
2     int u,v, comp;
3     bool bridge;
4 };
5 vector<edge> e;
6 void addEdge(int u, int v) {
7     G[u].pb(sz(e)), G[v].pb(sz(e));
8     e.pb((edge){u,v,-1,false});
9 }
10 //d[i]=id de la dfs
11 //b[i]=lowest id reachable from i
12 int d[MAXN], b[MAXN], t;
13 int nbc;//cant componentes
14 int comp[MAXN]; //comp[i]=cant comp biconexas a la cual pertenece i
15 void initDfs(int n) {
16     zero(G), zero(comp);
17     e.clear();
18     forn(i,n) d[i]=-1;
19     nbc = t = 0;
20 }
21 stack<int> st;
22 void dfs(int u, int pe) { //O(n + m)
23     b[u] = d[u] = t++;
24     comp[u] = (pe != -1);
25     forall(ne, G[u]) if (*ne != pe){
26         int v = e[*ne].u ^ e[*ne].v ^ u;
27         if (d[v] == -1) {
28             st.push(*ne);
29             dfs(v,*ne);
30             if (b[v] > d[u]){
31                 e[*ne].bridge = true; // bridge
32             }
33             if (b[v] >= d[u]){ // art
34                 int last;
35                 do {
36                     last = st.top(); st.pop();
37                     e[last].comp = nbc;

```

```

38     } while (last != *ne);
39     nbc++;
40     comp[u]++;
41 }
42 b[u] = min(b[u], b[v]);
43 }
44 else if (d[v] < d[u]) { // back edge
45     st.push(*ne);
46     b[u] = min(b[u], d[v]);
47 }
48 }
49 }

```

7.9. LCA + Climb

```

1 //f[v][k] holds the 2^k father of v
2 //L[v] holds the level of v
3 int N, f[100001][20], L[100001];
4 void build(){ //f[i][0] must be filled previously, O(nlgn)
5     forn(k, 20-1) forn(i, N) f[i][k+1]=f[f[i][k]][k];}
6
7 #define lg(x) (31-__builtin_clz(x))//=floor(log2(x))
8
9 int climb(int a, int d){ //O(lgn)
10     if(!d) return a;
11     dforn(i, lg(L[a])+1)
12         if(1<<i<=d)
13             a=f[a][i], d-=1<<i;
14     return a;
15 }
16 int lca(int a, int b){ //O(lgn)
17     if(L[a]<L[b]) swap(a, b);
18     a=climb(a, L[a]-L[b]);
19     if(a==b) return a;
20     dforn(i, lg(L[a])+1)
21         if(f[a][i]!=f[b][i])
22             a=f[a][i], b=f[b][i];
23     return f[a][0];
24 }

```

7.10. Heavy Light Decomposition

```

1 int treesz[MAXN]; //cantidad de nodos en el subarbol del nodo v
2 int dad[MAXN]; //dad[v]=padre del nodo v

```

```

3 void dfs1(int v, int p=-1){//pre-dfs
4     dad[v]=p;
5     treesz[v]=1;
6     forall(it, G[v]) if(*it!=p){
7         dfs1(*it, v);
8         treesz[v]+=treesz[*it];
9     }
10 }
11 int pos[MAXN], q;//pos[v]=posicion del nodo v en el recorrido de la dfs
12 //Las cadenas aparecen continuas en el recorrido!
13 int cantcad;
14 int homecad[MAXN];//dada una cadena devuelve su nodo inicial
15 int cad[MAXN];//cad[v]=cadena a la que pertenece el nodo
16 void heavylight(int v, int cur=-1){
17     if(cur==--1) homecad[cur=cantcad++]=v;
18     pos[v]=q++;
19     cad[v]=cur;
20     int mx=-1;
21     forn(i, sz(G[v])) if(G[v][i]!=dad[v])
22         if(mx==--1 || treesz[G[v][mx]]<treesz[G[v][i]]) mx=i;
23     if(mx!=--1) heavylight(G[v][mx], cur);
24     forn(i, sz(G[v])) if(i!=mx && G[v][i]!=dad[v])
25         heavylight(G[v][i], -1);
26 }
27 //ejemplo de obtener el maximo numero en el camino entre dos nodos
28 //RTA: max(query(low, u), query(low, v)), con low=lca(u, v)
29 //esta funcion va trepando por las cadenas
30 int query(int an, int v){//O(logn)
31     //si estan en la misma cadena:
32     if(cad[an]==cad[v]) return rmq.get(pos[an], pos[v]+1);
33     return max(query(an, dad[homecad[cad[v]]]),
34         rmq.get(pos[homecad[cad[v]]], pos[v]+1));
35 }

```

7.11. Euler Cycle

```

1 int n,m,ars[MAXE], eq;
2 vector<int> G[MAXN];//fill G,n,m,ars,eq
3 list<int> path;
4 int used[MAXN];
5 bool usede[MAXE];
6 queue<list<int>::iterator> q;
7 int get(int v){

```

```

8     while(used[v]<sz(G[v]) && usede[ G[v][used[v]] ]) used[v]++;
9     return used[v];
10 }
11 void explore(int v, int r, list<int>::iterator it){
12     int ar=G[v][get(v)]; int u=v^ars[ar];
13     usede[ar]=true;
14     list<int>::iterator it2=path.insert(it, u);
15     if(u!=r) explore(u, r, it2);
16     if(get(v)<sz(G[v])) q.push(it);
17 }
18 void euler(){
19     zero(used), zero(usede);
20     path.clear();
21     q=queue<list<int>::iterator>();
22     path.push_back(0); q.push(path.begin());
23     while(sz(q)){
24         list<int>::iterator it=q.front(); q.pop();
25         if(used[*it]<sz(G[*it])) explore(*it, *it, it);
26     }
27     reverse(path.begin(), path.end());
28 }
29 void addEdge(int u, int v){
30     G[u].pb(eq), G[v].pb(eq);
31     ars[eq++]=u^v;
32 }

```

7.12. Chu-liu

```

1 void visit(graph &h, int v, int s, int r,
2     vector<int> &no, vector< vector<int> > &comp,
3     vector<int> &prev, vector< vector<int> > &next, vector<weight> &mcost,
4     vector<int> &mark, weight &cost, bool &found) {
5     if (mark[v]) {
6         vector<int> temp = no;
7         found = true;
8         do {
9             cost += mcost[v];
10            v = prev[v];
11            if (v != s) {
12                while (comp[v].size() > 0) {
13                    no[comp[v].back()] = s;
14                    comp[s].push_back(comp[v].back());
15                    comp[v].pop_back();

```

```

16     }
17 }
18 } while (v != s);
19 forall(j, comp[s]) if (*j != r) forall(e, h[*j])
20     if (no[e->src] != s) e->w -= mcost[ temp[*j] ];
21 }
22 mark[v] = true;
23 forall(i, next[v]) if (no[*i] != no[v] && prev[no[*i]] == v)
24     if (!mark[no[*i]] || *i == s)
25         visit(h, *i, s, r, no, comp, prev, next, mcost, mark, cost, found)
26         ;
27 }
28 weight minimumSpanningArborescence(const graph &g, int r) {
29     const int n=sz(g);
30     graph h(n);
31     forn(u,n) forall(e,g[u]) h[e->dst].pb(*e);
32     vector<int> no(n);
33     vector<vector<int>> > comp(n);
34     forn(u, n) comp[u].pb(no[u] = u);
35     for (weight cost = 0; ;) {
36         vector<int> prev(n, -1);
37         vector<weight> mcost(n, INF);
38         forn(j,n) if (j != r) forall(e,h[j])
39             if (no[e->src] != no[j])
40                 if (e->w < mcost[ no[j] ])
41                     mcost[ no[j] ] = e->w, prev[ no[j] ] = no[e->src];
42         vector< vector<int> > next(n);
43         forn(u,n) if (prev[u] >= 0)
44             next[ prev[u] ].push_back(u);
45         bool stop = true;
46         vector<int> mark(n);
47         forn(u,n) if (u != r && !mark[u] && !comp[u].empty()) {
48             bool found = false;
49             visit(h, u, u, r, no, comp, prev, next, mcost, mark, cost, found);
50             if (found) stop = false;
51         }
52         if (stop) {
53             forn(u,n) if (prev[u] >= 0) cost += mcost[u];
54             return cost;
55         }
56     }
57 }

```

7.13. Hungarian

```

1 #define MAXN 256
2 #define INFTO 0x7f7f7f7f
3 int n;
4 int mt[MAXN][MAXN]; // Matriz de costos (X * Y)
5 int xy[MAXN], yx[MAXN]; // Matching resultante (X->Y, Y->X)
6 int lx[MAXN], ly[MAXN], slk[MAXN], slkx[MAXN], prv[MAXN];
7 char S[MAXN], T[MAXN];
8 void updtree(int x) {
9     forn(y, n) if (lx[x] + ly[y] - mt[x][y] < slk[y]) {
10         slk[y] = lx[x] + ly[y] - mt[x][y];
11         slkx[y] = x;
12     }
13 }
14 int hungar() { // Matching maximo de mayor costo en grafos dirigidos (N^3)
15     forn(i, n) {
16         ly[i] = 0;
17         lx[i] = *max_element(mt[i], mt[i]+n);
18     }
19     memset(xy, -1, sizeof(xy));
20     memset(yx, -1, sizeof(yx));
21     forn(m, n) {
22         memset(S, 0, sizeof(S));
23         memset(T, 0, sizeof(T));
24         memset(prv, -1, sizeof(prv));
25         memset(slk, 0x7f, sizeof(slk));
26         queue<int> q;
27         #define bpone(e, p) { q.push(e); prv[e] = p; S[e] = 1; updtree(e); }
28         forn(i, n) if (xy[i] == -1) { bpone(i, -2); break; }
29         int x=0, y=-1;
30         while (y!=-1) {
31             while (!q.empty() && y!=-1) {
32                 x = q.front(); q.pop();
33                 forn(j, n) if (mt[x][j] == lx[x] + ly[j] && !T[j]) {
34                     if (yx[j] == -1) { y = j; break; }
35                     T[j] = 1;
36                     bpone(yx[j], x);
37                 }
38             }
39             if (y!=-1) break;
40             int dlt = INFTO;
41             forn(j, n) if (!T[j]) dlt = min(dlt, slk[j]);
42             forn(k, n) {
43                 if (S[k]) lx[k] -= dlt;

```



```

42     if (T[k]) ly[k] += dlt;
43     if (!T[k]) slk[k] -= dlt;
44 }
45 forn(j, n) if (!T[j] && !slk[j]) {
46     if (yx[j] == -1) {
47         x = slkx[j]; y = j; break;
48     } else {
49         T[j] = 1;
50         if (!S[yx[j]]) bpone(yx[j], slkx[j]);
51     }
52 }
53 }
54 if (y!=-1) {
55     for(int p = x; p != -2; p = prv[p]) {
56         yx[y] = p;
57         int ty = xy[p]; xy[p] = y; y = ty;
58     }
59 } else break;
60 }
61 int res = 0;
62 forn(i, n) res += mt[i][xy[i]];
63 return res;
64 }

```

8. Network Flow

8.1. Dinic

```

1  int nodes, src, dest;
2  int dist[MAX], q[MAX], work[MAX];
3
4  struct Edge {
5      int to, rev;
6      ll f, cap;
7      Edge(int to, int rev, ll f, ll cap) : to(to), rev(rev), f(f), cap(cap)
8      {}
9  };
10 vector<Edge> G[MAX];
11
12 // Adds bidirectional edge
13 void addEdge(int s, int t, ll cap){
14     G[s].push_back(Edge(t, G[t].size(), 0, cap));

```

```

15     G[t].push_back(Edge(s, G[s].size()-1, 0, 0));
16 }
17
18 bool dinic_bfs() {
19     fill(dist, dist + nodes, -1);
20     dist[src] = 0;
21     int qt = 0;
22     q[qt++] = src;
23     for (int qh = 0; qh < qt; qh++) {
24         int u = q[qh];
25         forall(e, G[u]){
26             int v = e->to;
27             if(dist[v]<0 && e->f < e->cap){
28                 dist[v]=dist[u]+1;
29                 q[qt++]=v;
30             }
31         }
32     }
33     return dist[dest] >= 0;
34 }
35
36 ll dinic_dfs(int u, ll f) {
37     if (u == dest) return f;
38     for (int &i = work[u]; i < (int) G[u].size(); i++) {
39         Edge &e = G[u][i];
40         if (e.cap <= e.f) continue;
41         int v = e.to;
42         if (dist[v] == dist[u] + 1) {
43             ll df = dinic_dfs(v, min(f, e.cap - e.f));
44             if (df > 0) {
45                 e.f += df;
46                 G[v][e.rev].f -= df;
47                 return df;
48             }
49         }
50     }
51     return 0;
52 }
53
54 ll maxFlow(int _src, int _dest) { //O(V^2 E)
55     src = _src;
56     dest = _dest;
57     ll result = 0;

```

```

58 while (dinic_bfs()) {
59     fill(work, work + nodes, 0);
60     while(11 delta = dinic_dfs(src, INF))
61         result += delta;
62 }
63
64 // todos los nodos con dist[v]!=-1 vs los que tienen dist[v]==-1
65 // forman el min cut
66
67 return result;
68 }

```

8.2. Konig

```

1 // asume que el dinic YA ESTA tirado
2 // asume que nodes-1 y nodes-2 son la fuente y destino
3 int match[maxnodes]; // match[v]=u si u-v esta en el matching, -1 si v
4 // no esta matcheado
5 int s[maxnodes]; // numero de la bfs del koning
6 queue<int> kq;
7 // s[e]%2==1 o si e esta en V1 y s[e]==-1-> lo agarras
8 void koning() { // O(n)
9     for(v,nodes-2) s[v] = match[v] = -1;
10    for(v,nodes-2) forall(it,g[v]) if (it->to < nodes-2 && it->f>0)
11        { match[v]=it->to; match[it->to]=v;}
12    for(v,nodes-2) if (match[v]==-1) {s[v]=0;kq.push(v);}
13    while(!kq.empty()) {
14        int e = kq.front(); kq.pop();
15        if (s[e]%2==1) {
16            s[match[e]] = s[e]+1;
17            kq.push(match[e]);
18        } else {
19            forall(it,g[e]) if (it->to < nodes-2 && s[it->to]==-1) {
20                s[it->to] = s[e]+1;
21                kq.push(it->to);
22            }
23        }
24    }
25 }

```

8.3. Edmonds Karp's

```

1 #define MAX_V 1000

```

```

2 #define INF 1e9
3 //special nodes
4 #define SRC 0
5 #define SNK 1
6 map<int, int> G[MAX_V]; //limpiar esto
7 //To add an edge use
8 #define add(a, b, w) G[a][b]=w
9 int f, p[MAX_V];
10 void augment(int v, int minE){
11     if(v==SRC) f=minE;
12     else if(p[v]!=-1){
13         augment(p[v], min(minE, G[p[v]][v]));
14         G[p[v]][v]-=f, G[v][p[v]]+=f;
15     }
16 }
17 ll maxflow(){ //O(VE^2)
18     ll Mf=0;
19     do{
20         f=0;
21         char used[MAX_V]; queue<int> q; q.push(SRC);
22         zero(used), memset(p, -1, sizeof(p));
23         while(sz(q)){
24             int u=q.front(); q.pop();
25             if(u==SNK) break;
26             forall(it, G[u])
27                 if(it->snd>0 && !used[it->fst])
28                     used[it->fst]=true, q.push(it->fst), p[it->fst]=u;
29         }
30         augment(SNK, INF);
31         Mf+=f;
32     }while(f);
33     return Mf;
34 }

```

8.4. Push-Relabel

```

1 #define MAX_V 1000
2 int N; //valid nodes are [0...N-1]
3 #define INF 1e9
4 //special nodes
5 #define SRC 0
6 #define SNK 1
7 map<int, int> G[MAX_V];

```

```

8 //To add an edge use
9 #define add(a, b, w) G[a][b]=w
10 ll excess[MAX_V];
11 int height[MAX_V], active[MAX_V], count[2*MAX_V+1];
12 queue<int> Q;
13 void enqueue(int v) {
14     if (!active[v] && excess[v] > 0) active[v]=true, Q.push(v); }
15 void push(int a, int b) {
16     int amt = min(excess[a], ll(G[a][b]));
17     if(height[a] <= height[b] || amt == 0) return;
18     G[a][b]-=amt, G[b][a]+=amt;
19     excess[b] += amt, excess[a] -= amt;
20     enqueue(b);
21 }
22 void gap(int k) {
23     forn(v, N){
24         if (height[v] < k) continue;
25         count[height[v]]--;
26         height[v] = max(height[v], N+1);
27         count[height[v]]++;
28         enqueue(v);
29     }
30 }
31 void relabel(int v) {
32     count[height[v]]--;
33     height[v] = 2*N;
34     forall(it, G[v])
35         if(it->snd)
36             height[v] = min(height[v], height[it->fst] + 1);
37     count[height[v]]++;
38     enqueue(v);
39 }
40 ll maxflow() { //O(V^3)
41     zero(height), zero(active), zero(count), zero(excess);
42     count[0] = N-1;
43     count[N] = 1;
44     height[SRC] = N;
45     active[SRC] = active[SNK] = true;
46     forall(it, G[SRC]){
47         excess[SRC] += it->snd;
48         push(SRC, it->fst);
49     }
50     while(sz(Q)) {

```

```

51         int v = Q.front(); Q.pop();
52         active[v]=false;
53         forall(it, G[v]) push(v, it->fst);
54         if(excess[v] > 0)
55             count[height[v]] == 1? gap(height[v]):relabel(v);
56     }
57     ll mf=0;
58     forall(it, G[SRC]) mf+=G[it->fst][SRC];
59     return mf;
60 }

```

8.5. Min-cost Max-flow

```

1 struct edge {
2     int u, v;
3     ll cap, cost, flow;
4     ll rem() { return cap - flow; }
5 };
6 int n; //numero de nodos
7 vector<int> G[MAXN];
8 vector<edge> e;
9 void addEdge(int u, int v, ll cap, ll cost) {
10     G[u].pb(si(e)); e.pb((edge){u,v,cap,cost,0});
11     G[v].pb(si(e)); e.pb((edge){v,u,0,-cost,0});
12 }
13 ll pot[MAXN], dist[MAXN], pre[MAXN], cap[MAXN];
14 ll mxFlow, mnCost;
15 void flow(int s, int t) {
16     fill(pot, pot+n, 0);
17     mxFlow=mnCost=0;
18     while(1){
19         fill(dist, dist+n, INF); dist[s] = 0;
20         fill(pre, pre+n, -1); pre[s]=0;
21         fill(cap, cap+n, 0); cap[s] = INF;
22         priority_queue<pair<ll,int> > q; q.push(mkp(0,s));
23         while (!q.empty()) {
24             pair<ll,int> top = q.top(); q.pop();
25             int u = top.second, d = -top.first;
26             if (u == t) break;
27             if (d > dist[u]) continue;
28             forn(i, si(G[u])) {
29                 edge E = e[G[u][i]];
30                 int c = E.cost + pot[u] - pot[E.v];

```

```

31     if (E.rem() && dist[E.v] > dist[u] + c) {
32         dist[E.v] = dist[u] + c;
33         pre[E.v] = G[u][i];
34         cap[E.v] = min(cap[u], E.rem());
35         q.push(mkp(-dist[E.v], E.v));
36     }
37 }
38 }
39 if (pre[t] == -1) break;
40 forn(u,n)
41     if (dist[u] == INF) pot[u] = INF;
42     else pot[u] += dist[u];
43 mxFlow +=cap[t];
44 mnCost +=cap[t]*pot[t];
45 for (int v = t; v != s; v = e[pre[v]].u) {
46     e[pre[v]].flow += cap[t];
47     e[pre[v]^1].flow -= cap[t];
48 }
49 }
50 }

```

9. Ayudamemoria

Cant. decimales

```

1 #include <iomanip>
2 cout << setprecision(2) << fixed;

```

Rellenar con espacios(para justificar)

```

1 #include <iomanip>
2 cout << setfill('␣') << setw(3) << 2 << endl;

```

Leer hasta fin de linea

```

1 #include <sstream>
2 //hacer cin.ignore() antes de getline()
3 while(getline(cin, line)){
4     istringstream is(line);
5     while(is >> X)
6         cout << X << "␣";
7     cout << endl;
8 }

```

Aleatorios

```

1 #define RAND(a, b) (rand()%(b-a+1)+a)
2 srand(time(NULL));

```

Doubles Comp.

```

1 const double EPS = 1e-9;
2 x == y <=> fabs(x-y) < EPS
3 x > y <=> x > y + EPS
4 x >= y <=> x > y - EPS

```

Limites

```

1 #include <limits>
2 numeric_limits<T>
3     ::max()
4     ::min()
5     ::epsilon()

```

Muahaha

```

1 #include <signal.h>
2 void divzero(int p){
3     while(true);}
4 void segm(int p){
5     exit(0);}
6 //in main
7 signal(SIGFPE, divzero);
8 signal(SIGSEGV, segm);

```

Mejorar velocidad

```

1 ios::sync_with_stdio(false);

```

Mejorar velocidad 2

```

1 //Solo para enteros positivos
2 inline void Scanf(int& a){
3     char c = 0;
4     while(c<33) c = getc(stdin);
5     a = 0;
6     while(c>33) a = a*10 + c - '0', c = getc(stdin);
7 }

```

Leer del teclado

```
1 | freopen("/dev/tty", "a", stdin);
```

Iterar subconjunto

```
1 | for(int sbm=bm; sbm; sbm=(sbm-1)&bm)
```

File setup

```
1 | //tambien se pueden usar comas: {a, x, m, l}  
2 | touch {a..l}.in; tee {a..l}.cpp < template.cpp
```